

Optimalizace

Jiří Filipovič

podzim 2010

Naivní implementace

```
__global__ void mmul(float *A, float *B, float *C, int n){
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.y + threadIdx.y;

    float tmp = 0;
    for (int k = 0; k < n; k++)
        tmp += A[y*n+k] * B[k*n+x];

    C[y*n + x] = tmp;
}
```

Co jsme se naučili

Naivní implementace algoritmu

- každý thread zpracovává odděleně jeden element výsledné matice
- omezena propustností paměti
- teoretické maximum jsme určili jako 66.8 GFlops
- výkon velmi závislý na uspořádání threadů – bloky 128×1 dávají výkon 36.6 GFlops, bloky 1×128 3.9 GFlops

Co jsme se naučili

Naivní implementace algoritmu

- každý thread zpracovává odděleně jeden element výsledné matice
- omezena propustností paměti
- teoretické maximum jsme určili jako 66.8 GFlops
- výkon velmi závislý na uspořádání threadů – bloky 128×1 dávají výkon 36.6 GFlops, bloky 1×128 3.9 GFlops

Nyní rozumíme rozdílným výsledkům

- teoretického maxima nelze docílit – z paměti GPU přenášíme po nejméně 32-bytových částech, musíme tedy přenést více dat, než je nutné
- je-li 128 threadů v bloku zarovnáno ve směru osy x , je přenos dat neprokládaný, v opačném případě je prokládaný

Co jsme se naučili

Navrhli jsme blokovou implementaci

- každý blok threadů načítá bloky matic A a B do sdílené paměti, znovuužívá data ke snížení omezení přenosovou rychlostí globální paměti
- teoretické maximum 568 GFlops, dosáhli jsme 198 GFlops

S novými znalostmi můžeme jeho implementaci přehodnotit...

Násobení po blocích

```

__global__ void mmul(float *A, float *B, float *C, int n){
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    __shared__ float As[BLOCK][BLOCK];
    __shared__ float Bs[BLOCK][BLOCK];

    float Csub = 0.0f;
    for (int b = 0; b < n/BLOCK; b++){
        As[ty][tx] = A[(ty + by*BLOCK)*n + b*BLOCK+tx];
        Bs[ty][tx] = B[(ty + b*BLOCK)*n + bx*BLOCK+tx];
        __syncthreads();

        for (int k = 0; k < BLOCK; k++)
            Csub += As[ty][k]*Bs[k][tx];
        __syncthreads();
    }

    C[(ty + by*BLOCK)*n + bx*BLOCK+tx] = Csub;
}

```

Hříchy implementace

```
As[ty][tx] = A[(ty + by*BLOCK)*n + b*BLOCK+tx];
Bs[ty][tx] = B[(ty + b*BLOCK)*n + bx*BLOCK+tx];
...
C[(ty + by*BLOCK)*n + bx*BLOCK+tx] = Csub;
```

Přístup do globální paměti se zdá být v pořádku.

```
Csub += As[ty][k]*Bs[k][tx];
```

Přístup do sdílené také

- má-li blok threadů velikost ve smyslu osy x násobek velikosti warpu, dochází u proměnné As k broadcastu
- proměnná Bs je čtena v souvislých řádcích, přístup tedy negeneruje konflikty bank

Teoretické maximum

Lze určit přesněji teoretické omezení výkonu?

- maximum jsme určili podle výkonu GPU v MAD instrukcích (622 GFlops)
- nyní víme, že MAD instrukce pracující s operandem ve sdílené paměti pracují rychlostí 6 taktů na warp
- nově lze tedy teoretické maximum určit jako 415 GFlops
- stále jsme od něj však daleko

Ztráty výkonu

Co nás vzdaluje od maxima?

- overhead spuštění kernelu a spouštění threadů
 - z principu se mu nevyhneme, počet threadů lze redukovat
- operace „režije“
 - pointerová aritmetika, cykly
 - lze redukovat
- synchronizace
 - může a nemusí být problém
- load/store ve výpočtu
 - dva operandy v SMEM na jednu MAD instrukci
 - je tedy zapotřebí jeden load na jednu MAD

Počítáme-li výkonový strop pro kombinaci load + MAD s operandem ve sdílené paměti, dostaneme se k omezení 244 GFlops.

- od toho již nejsou naměřené výsledky příliš vzdáleny

Nalezení lepší implementace

Lze počet load instrukcí omezit?

Nalezení lepší implementace

Lze počet load instrukcí omezit?

- data ve sdílené paměti snižují přenosy z paměti globální

Nalezení lepší implementace

Lze počet load instrukcí omezit?

- data ve sdílené paměti snižují přenosy z paměti globální
- můžeme snížit přenosy ze sdílené paměti pomocí dat v registrech?

Nalezení lepší implementace

Lze počet load instrukcí omezit?

- data ve sdílené paměti snižují přenosy z paměti globální
- můžeme snížit přenosy ze sdílené paměti pomocí dat v registrech?
- můžeme – stačí nechat pracovat méně threadů nad více daty

Nalezení lepší implementace

Lze počet load instrukcí omezit?

- data ve sdílené paměti snižují přenosy z paměti globální
- můžeme snížit přenosy ze sdílené paměti pomocí dat v registrech?
- můžeme – stačí nechat pracovat méně threadů nad více daty

Blok o velikosti $m \times n$ threadů necháme pracovat s daty o velikosti $m \times m$, kde $m = n \cdot k$; $k \in \mathbb{N}$.

- větší bloky potenciálně nevýhodné kvůli synchronizaci
- menší bloky potenciálně nevýhodné kvůli overheadu daném pointerovou aritmetikou
- experimentálně najdeme vhodnou velikost bloku

Nalezení lepší implementace

Nejlepší výsledky dosaženy pro bloky velikosti 32×32 , na kterých pracuje 32×16 threadů.

- půl loadu na jednu MAD instrukci dává teoretické omezení 311 GFlops
- naměřili jsme 235.4 GFlops
- něco je ještě špatně

Deassembling kódu

Zaměříme se na vnitřní smyčku

```
Csub1 += As[ty][k]*Bs[k][tx];
Csub2 += As[ty+16][k]*Bs[k][tx];
```

...

```
mov.b32 $r0, s[$ofs4+0x0000]
add.b32 $ofs4, $ofs2, 0x00000180
mad.rn.f32 $r7, s[$ofs1+0x0008], $r0, $r7
mad.rn.f32 $r8, s[$ofs3+0x0008], $r0, $r8
```

...

Deassembling kódu

Zaměříme se na vnitřní smyčku

```
Csub1 += As[ty][k]*Bs[k][tx];
Csub2 += As[ty+16][k]*Bs[k][tx];
```

...

```
mov.b32 $r0, s[$ofs4+0x0000]
add.b32 $ofs4, $ofs2, 0x00000180
mad.rn.f32 $r7, s[$ofs1+0x0008], $r0, $r7
mad.rn.f32 $r8, s[$ofs3+0x0008], $r0, $r8
```

...

Kompilátor dokázal převést adresaci přes k na konstantní offsety pouze u proměnné As

- k Bs je přístupováno prokládaně
- znamená to jednu add instrukci navíc

Odstranění add instrukce

Do pole Bs můžeme ukládat transponovaná data, pak vypadá kód vnitřní smyčky takto

```
Csub1 += As[ty][k]*Bs[tx][k];  
Csub2 += As[ty+16][k]*Bs[tx][k];
```

Odstranění add instrukce

Do pole Bs můžeme ukládat transponovaná data, pak vypadá kód vnitřní smyčky takto

```
Csub1 += As[ty][k]*Bs[tx][k];
Csub2 += As[ty+16][k]*Bs[tx][k];
```

Ve výsledném assembleru již instrukce add chybí

```
...
mov.b32 $r0, s[$ofs4+0x0008]
mad.rn.f32 $r6, s[$ofs3+0x0034], $r0, $r6
mad.rn.f32 $r8, s[$ofs1+0x0008], $r0, $r8
...
```

Odstranění add instrukce

Do pole *Bs* můžeme ukládat transponovaná data, pak vypadá kód vnitřní smyčky takto

```
Csub1 += As[ty][k]*Bs[tx][k];
Csub2 += As[ty+16][k]*Bs[tx][k];
```

Ve výsledném assembleru již instrukce `add` chybí

```
...
mov.b32 $r0, s[$ofs4+0x0008]
mad.rn.f32 $r6, s[$ofs3+0x0034], $r0, $r6
mad.rn.f32 $r8, s[$ofs1+0x0008], $r0, $r8
...
```

Nový problém – konflikty bank sdílené paměti

Odstranění add instrukce

Do pole *Bs* můžeme ukládat transponovaná data, pak vypadá kód vnitřní smyčky takto

```
Csub1 += As[ty][k]*Bs[tx][k];
Csub2 += As[ty+16][k]*Bs[tx][k];
```

Ve výsledném assembleru již instrukce `add` chybí

```
...
mov.b32 $r0, s[$ofs4+0x0008]
mad.rn.f32 $r6, s[$ofs3+0x0034], $r0, $r6
mad.rn.f32 $r8, s[$ofs1+0x0008], $r0, $r8
...
```

Nový problém – konflikty bank sdílené paměti

- vyřeší padding

Odstranění add instrukce

Do pole *Bs* můžeme ukládat transponovaná data, pak vypadá kód vnitřní smyčky takto

```
Csub1 += As[ty][k]*Bs[tx][k];
Csub2 += As[ty+16][k]*Bs[tx][k];
```

Ve výsledném assembleru již instrukce `add` chybí

```
...
mov.b32 $r0, s[$ofs4+0x0008]
mad.rn.f32 $r6, s[$ofs3+0x0034], $r0, $r6
mad.rn.f32 $r8, s[$ofs1+0x0008], $r0, $r8
...
```

Nový problém – konflikty bank sdílené paměti

- vyřeší padding

Výsledná rychlost: 276.2 GFlops.

Lze matice násobit ještě rychleji?

Naměřený výkon je již poměrně blízký teoretickému maximu

- rozdíl je dán spouštěním kernelu/threadů, synchronizací a pointerovou aritmetikou
- chceme-li dosáhnout vyšší rychlosti, je třeba přehodnotit algoritmus

Lze matice násobit ještě rychleji?

Naměřený výkon je již poměrně blízký teoretickému maximu

- rozdíl je dán spouštěním kernelu/threadů, synchronizací a pointerovou aritmetikou
- chceme-li dosáhnout vyšší rychlosti, je třeba přehodnotit algoritmus

Zásadním problémem je, že spolu násobíme dvě matice ve sdílené paměti

- nutnost provádět load instrukce spolu s MAD instrukcemi

Lze matice násobit ještě rychleji?

Naměřený výkon je již poměrně blízky teoretickému maximu

- rozdíl je dán spouštěním kernelu/threadů, synchronizací a pointerovou aritmetikou
- chceme-li dosáhnout vyšší rychlosti, je třeba přehodnotit algoritmus

Zásadním problémem je, že spolu násobíme dvě matice ve sdílené paměti

- nutnost provádět load instrukce spolu s MAD instrukcemi

Můžeme mít ve sdílené paměti jen jeden blok?

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

- provádíme iterativně rank-1 update bloků v C ze sloupce matice A a řádku matice B

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

- provádíme iterativně rank-1 update bloků v C ze sloupce matice A a řádku matice B
- sloupce je nutno číst se sdílené paměti

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

- provádíme iterativně rank-1 update bloků v C ze sloupce matice A a řádku matice B
- sloupce je nutno číst se sdílené paměti
- řádky můžeme načítat postupně, lze tedy použít data v registrech

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

- provádíme iterativně rank-1 update bloků v C ze sloupce matice A a řádku matice B
- sloupce je nutno číst se sdílené paměti
- řádky můžeme načítat postupně, lze tedy použít data v registrech
- výsledný blok může být uložen v registrech

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

- provádíme iterativně rank-1 update bloků v C ze sloupce matice A a řádku matice B
- sloupce je nutno číst se sdílené paměti
- řádky můžeme načítat postupně, lze tedy použít data v registrech
- výsledný blok může být uložen v registrech
- pracujeme tedy pouze s jedním operandem ve sdílené paměti, není nutný load

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

- provádíme iterativně rank-1 update bloků v C ze sloupce matice A a řádku matice B
- sloupce je nutno číst se sdílené paměti
- řádky můžeme načítat postupně, lze tedy použít data v registrech
- výsledný blok může být uložen v registrech
- pracujeme tedy pouze s jedním operandem ve sdílené paměti, není nutný load
- není nutná aritmetika uprostřed smyčky (viz předchozí optimalizace)

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

- provádíme iterativně rank-1 update bloků v C ze sloupce matice A a řádku matice B
- sloupce je nutno číst se sdílené paměti
- řádky můžeme načítat postupně, lze tedy použít data v registrech
- výsledný blok může být uložen v registrech
- pracujeme tedy pouze s jedním operandem ve sdílené paměti, není nutný load
- není nutná aritmetika uprostřed smyčky (viz předchozí optimalizace)
- teoretické maximum výkonu je tak omezeno rychlostí instrukce MAD pracující se sdílenou pamětí na cca 415 GFlops

Implementace

Nejvyšší rychlosti bylo dosaženo s konfigurací

- matice A zpracovávána po blocích 16×16 , uložených ve sdílené paměti
- matice B zpracovávána po blocích 64×1 , uložených v registrech
- bloky matice C mají tedy rozměr 64×16 , jsou uloženy v registrech

Implementace

Nejvyšší rychlosti bylo dosaženo s konfigurací

- matice A zpracovávána po blocích 16×16 , uložených ve sdílené paměti
- matice B zpracovávána po blocích 64×1 , uložených v registrech
- bloky matice C mají tedy rozměr 64×16 , jsou uloženy v registrech

Dosažená rychlost této implementace **375 GFlops**.

Shrnutí

Implementace	rychlost	rel. Δ	abs. Δ
Naivní implementace, thready 1×128	3.9 GFlops		
Naivní implementace	36.6 GFlops	9.4 \times	9.4 \times
Blokový přístup	198 GFlops	5.4 \times	51 \times
Bloky 32×16 pracující s daty 32×16	235 GFlops	1.19 \times	60 \times
Odstranění ADD instrukce	276 GFlops	1.17 \times	71 \times
Jen jeden blok ve sdílené paměti	375 GFlops	1.36 \times	96 \times

Shrnutí

Implementace	rychlost	rel. Δ	abs. Δ
Naivní implementace, thready 1×128	3.9 GFlops		
Naivní implementace	36.6 GFlops	9.4 \times	9.4 \times
Blokový přístup	198 GFlops	5.4 \times	51 \times
Bloky 32×16 pracující s daty 32×16	235 GFlops	1.19 \times	60 \times
Odstranění ADD instrukce	276 GFlops	1.17 \times	71 \times
Jen jeden blok ve sdílené paměti	375 GFlops	1.36 \times	96 \times

- Nejzásadnější je redukce poměru aritmetických operací k pamětovým přenosům a základní optimalizace přístupu do paměti.

Shrnutí

Implementace	rychlost	rel. Δ	abs. Δ
Naivní implementace, thready 1×128	3.9 GFlops		
Naivní implementace	36.6 GFlops	9.4 \times	9.4 \times
Blokový přístup	198 GFlops	5.4 \times	51 \times
Bloky 32×16 pracující s daty 32×16	235 GFlops	1.19 \times	60 \times
Odstranění ADD instrukce	276 GFlops	1.17 \times	71 \times
Jen jeden blok ve sdílené paměti	375 GFlops	1.36 \times	96 \times

- Nejzásadnější je redukce poměru aritmetických operací k pamětovým přenosům a základní optimalizace přístupu do paměti.
- Optimalizace na úrovni instrukcí je relativně náročná, avšak pro kritické kódy může přinést relativně významné zrychlení.

Součet prvků vektoru

Pro vektor v o n prvcích chceme spočítat $x = \sum_{i=1}^n v_i$.

Součet prvků vektoru

Pro vektor v o n prvcích chceme spočítat $x = \sum_{i=1}^n v_i$.
Zápis v jazyce C

```
int x = 0;
for (int i = 0; i < n; i++)
    x += v[i];
```

Jednotlivé iterace cyklu jsou na sobě závislé.

Součet prvků vektoru

Pro vektor v o n prvcích chceme spočítat $x = \sum_{i=1}^n v_i$.
Zápis v jazyce C

```
int x = 0;
for (int i = 0; i < n; i++)
    x += v[i];
```

Jednotlivé iterace cyklu jsou na sobě závislé.

- nemůžeme udělat všechnu práci paralelně
- sčítání je však (alespoň teoreticky :-)) asociativní
- není tedy nutno počítat sekvenčně

Paralelní algoritmus

Představený sekvenční algoritmus provádí pro 8 prvků výpočet:

$$(((((((v_1 + v_2) + v_3) + v_4) + v_5) + v_6) + v_7) + v_8$$

Paralelní algoritmus

Představený sekvenční algoritmus provádí pro 8 prvků výpočet:

$$(((((((v_1 + v_2) + v_3) + v_4) + v_5) + v_6) + v_7) + v_8$$

Sčítání je asociativní... spřeházejme tedy závorky:

$$((v_1 + v_2) + (v_3 + v_4)) + ((v_5 + v_6) + (v_7 + v_8))$$

Paralelní algoritmus

Představený sekvenční algoritmus provádí pro 8 prvků výpočet:

$$(((((((v_1 + v_2) + v_3) + v_4) + v_5) + v_6) + v_7) + v_8)$$

Sčítání je asociativní... spřeházejme tedy závorky:

$$((v_1 + v_2) + (v_3 + v_4)) + ((v_5 + v_6) + (v_7 + v_8))$$

Nyní můžeme pracovat paralelně

- v prvním kroku provedeme 4 sčítání
- ve druhém dvě
- ve třetím jedno

Celkově stejné množství práce ($n - 1$ sčítání), ale v $\log_2 n$ paralelních krocích!

Paralelní algoritmus

Našli jsme vhodný paralelní algoritmus

- provádí stejné množství operací jako sériová verze
- při dostatku procesorů je proveden v logaritmickém čase

Sčítáme výsledky předešlých součtů

- předešlé součty provádělo více threadů
- vyžaduje globální bariéru

Naivní přístup

Nejjednodušší schéma algoritmu:

- kernel pro sudá $i < n$ provede $v[i] += v[i+1]$
- opakujeme pro $n \neq 2$ dokud $n > 1$

Omezení výkonu

- $2n$ čtení z globální paměti
- n zápisů do globální paměti
- $\log_2 n$ volání kernelu

Na jednu aritmetickou operaci připadají 3 paměťové přenosy, navíc je nepříjemný overhead spouštění kernelu.

Využití rychlejší paměti

V rámci volání kernelu můžeme počítat více, než jen dvojice

- každý blok bx načte m prvků do sdílené paměti
- provede redukci (ve sdílené paměti v $\log_2 m$ krocích)
- uloží pouze jedno číslo odpovídající $\sum_{i=m \cdot bx}^{m \cdot bx + m} v_i$

Výhodnější z hlediska paměťových přenosů i spouštění kernelů

- $n + \frac{n}{m} + \frac{n}{m^2} + \dots + \frac{n}{m^{\log_m n}} = (n - 1) \frac{m}{m-1}$
- přibližně $n + \frac{n}{m}$ čtení, $\frac{n}{m}$ zápisů
- $\log_m n$ spuštění kernelu

Implementace 1

```
__global__ void reduce1(int *v){
    extern __shared__ int sv[];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sv[tid] = v[i];
    __syncthreads();

    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0)
            sv[tid] += sv[tid + s];
        __syncthreads();
    }

    if (tid == 0)
        v[blockIdx.x] = sv[0];
}
```


Vysoká úroveň divergence

- první iteraci pracuje každý 2. thread
- druhou iteraci pracuje každý 4. thread
- třetí iteraci pracuje každý 8 thread
- atd.

Přenos (GTX 280) 3.77 GB/s, 0.94 MElem/s.

Implementace 2

Nahradíme indexaci ve for cyklu

```
for (unsigned int s = 1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x)
        sv[index] += sv[index + s];
    __syncthreads();
}
```

Přenos 8.33 GB/s, 2.08 MElem/s.

Řeší divergenci, generuje konflikty bank.

Implementace 3

Tak ještě jinak...

```
for (unsigned int s = blockDim.x/2; s > 0; s >>= 1) {
    if (tid < s)
        sv[tid] += sv[tid + s];
    __syncthreads();
}
```

Žádná divergence ani konflikty.

Přenos 16.34 GB/s, 4.08 MElem/s.

Polovina threadů nic nepočítá...

Implementace 4

První sčítání provedeme již během načítání.

```
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;  
sv[tid] = v[i] + v[i+blockDim.x];
```

Přenos 27.16 GB/s, 6.79 MElem/s.

Data zřejmě čteme optimálně, stále je zde však výkonová rezerva – zaměřme se na instrukce.

Implementace 5

V jednotlivých krocích redukce ubývá aktivních threadů

- nakonec bude pracovat pouze jeden warp
- ten je však synchronizován implicitně, můžeme tedy odebrat `__syncthreads()`
- podmínka `if(tid < s)` je zde zbytečná (nic neušetří)

Unrollujme tedy poslední warp...

Implementace 5

```

for (unsigned int s = blockDim.x/2; s > 32; s >>= 1){
    if (tid < s)
        sv[tid] += sv[tid + s];
    __syncthreads();
}

if (tid < 32){
    sv[tid] += sv[tid + 32];
    sv[tid] += sv[tid + 16];
    sv[tid] += sv[tid + 8];
    sv[tid] += sv[tid + 4];
    sv[tid] += sv[tid + 2];
    sv[tid] += sv[tid + 1];
}

```

Ušetříme čas i ostatním waprům (zkončí dříve s for cyklem).
 Přenos 37.68 GB/s, 9.42 MElem/s.

Implementace 6

Jak je to s rozvinutím for cyklu?

Známe-li počet iterací, můžeme cyklus rozvinout

- počet iterací je závislý na velikosti bloku

Můžeme být obecní?

- algoritmus pracuje s bloky o velikosti 2^n
- velikost bloku je shora omezena
- známe-li při kompilaci velikost bloku, můžeme použít šablonu

```
template <unsigned int blockSize>
__global__ void reduce6(int *v)
```

Implementace 6

Podmínky s *blockSize* se vyhodnotí již při překladu:

```
if (blockSize >= 512){
    if (tid < 256)
        sv[tid] += sv[tid + 256];
    __syncthreads();
}
if (blockSize >= 256){
    if (tid < 128)
        sv[tid] += sv[tid + 128];
    __syncthreads();
}
if (blockSize >= 128){
    if (tid < 64)
        sv[tid] += sv[tid + 64];
    __syncthreads();
}
```


Implementace 6

```

if (tid < 32){
    if (blockSize >= 64) sv[tid] += sv[tid + 32];
    if (blockSize >= 32) sv[tid] += sv[tid + 16];
    if (blockSize >= 16) sv[tid] += sv[tid + 8];
    if (blockSize >= 8) sv[tid] += sv[tid + 4];
    if (blockSize >= 4) sv[tid] += sv[tid + 2];
    if (blockSize >= 2) sv[tid] += sv[tid + 1];
}

```

Spuštění kernelu:

```
reduce6<block><<<grid, block, mem>>>(d_v);
```

Přenos 50.64 GB/s, 12.66 MElem/s.

Implementace 7

Můžeme algoritmus ještě vylepšit?

Vraťme se zpět ke složitosti:

- celkem $\log n$ kroků
- celkem $n - 1$ sčítání
- časová složitost pro p threadů běžících paralelně (p procesorů)
 $\mathcal{O}\left(\frac{n}{p} + \log n\right)$

Cena paralelního výpočtu

- definována jako počet procesorů krát časová složitost
- přidělíme-li každému datovému elementu jeden thread, lze uvažovat $p = n$
- pak je cena $\mathcal{O}(n \cdot \log n)$
- není efektivní

Implementace 7

Snížení ceny

- použijeme $\mathcal{O}\left(\frac{n}{\log n}\right)$ threadů
- každý thread provede $\mathcal{O}(\log n)$ sekvenčních kroků
- následně se provede $\mathcal{O}(\log n)$ paralelních kroků
- časová složitost zůstane
- cena se sníží na $\mathcal{O}(n)$

Co to znamená v praxi?

- redukuje práci spojenou s vytvářením threadu a pointerovou aritmetikou
- to přináší výhodu v momentě, kdy máme výrazně více threadů, než je třeba k saturaci GPU
- navíc snižujeme overhead spouštění kernelů

Implementace 7

Modifikujeme načítání do sdílené paměti

```
unsigned int gridSize = blockSize*2*gridDim.x;
sv[tid] = 0;

while(i < n){
    sv[tid] += v[i] + v[i+blockSize];
    i += gridSize;
}
__syncthreads();
```

Přenos 77.21 GB/s, 19.3 MElem/s.

Implementace 7

Modifikujeme načítání do sdílené paměti

```
unsigned int gridSize = blockSize*2*gridDim.x;
sv[tid] = 0;

while(i < n){
    sv[tid] += v[i] + v[i+blockSize];
    i += gridSize;
}
__syncthreads();
```

Přenos 77.21 GB/s, 19.3 MElem/s.

Jednotlivé implementace jsou k nalezení v CUDA SDK.

Výběr vhodného problému

Než se pustíme do GPU akcelerace, je vhodné se zamyslet, jestli nám může pomoci :-).

Akcelerovaný problém by měl být

- kritický pro výkon aplikace
- musí se jednat o dostatečně velký problém (z hlediska počtu operací k jeho vyřešení)
- musí být paralelizovatelný (to zpravidla velké problémy jsou)
- k řešení problému musí být zapotřebí dostatek operací na jeden datový element (omezení přenosu zadání po PCI-E)

Optimalizujeme čas či spotřebu?

Postup návrhu algoritmu

Paralelizace

- v řešeném problému je třeba najít paralelismus
- již zde je vhodné uvažovat o omezeních architektury

Teoretické maximum rychlosti algoritmu

- než začneme implementovat, je vhodné mít představu, jak rychle může algoritmus na daném HW pracovat
- základní omezení dává paměťová propustnost a aritmetický výkon
- výkon CPU a GPU se může sčítat

Optimalizace

Je rozumné postupovat od obecně významnějších k méně významným (tak se jejich efekt lépe projeví)

- přístup do globální paměti (bandwidth, latence)
- přístup do ostatních pamětí
- konfigurace běhu (počet threadů na blok, množství práce na thread)
- divergence běhu
- optimalizace na úrovni instrukcí

Nezapomínejte

- poctivě benchmarkovat
- kontrolovat kód profilerem

Pozor na interpretaci rychlosti algoritmu

Efekt některých optimalizací může být skryt významnějšími neoptimalitami

- omezíme přednostním aplikováním významnějších optimalizací
- omezíme používáním profileru

Prostor optimalizací je nespojitý

- dáno omezenými zdroji GPU
- rychlejší kód threadu může vézt k celkově nižšímu výkonu

Výkon je závislý na velikosti problému

- menší instance mohou mít jiné nároky
- partition camping

Jaké zrychlení oproti CPU je reálné?

- základní odhad zrychlení vychází z porovnání aritmetického výkonu a propustnosti paměti
- GPU však nemusí přinést adekvátní zrychlení
 - nedostatečně či nevhodně paralelizovatelný algoritmus
 - nevhodné datové struktury, náhodný přístup
 - PCI-E bottleneck (málo výpočtu vzhledem k přenosům, multi-GPU algoritmy)
- GPU také může přinést vyšší zrychlení
 - významné využití SFU
 - komplikovaná vektorizace u CPU
 - degradace výkonu paměti u CPU
 - špatně škálovací SMP
- odlišné škálování CPU a GPU s rostoucí velikostí problému

Pozor na příliš optimistická měření

- vysoké zrychlení má často za příčinu špatný CPU algoritmus
- je zapotřebí si uvědomit, že CPU má více jader a vektorové jednotky
 - nevektorizovaný jednothreadový kód využívá (v jednoduché přesnosti) 1/16 teoretického maxima 4-jádrového CPU
- přínos GPU řešení lze podložit spočítáním flopsů

Prevence chyb

- testujte úspěšnost volání API a kernelů
 - chyby se jinak umí projevit se spožděním
- na GPU je docela deterministická alokace paměti
 - pokud se nic nezapíše, obvykle dostanete správný výsledek
 - pro účely ladění mažte výstupní data
- pozor na pošlapání sdílené paměti
 - kernel často nespadne, objevují se interference mezi bloky

Ladění výkonu

- je dobré psát snadno konfigurovatelný kód
 - konfigurovatelná velikost bloku, práce na thread, ...
 - konfigurace výhodnější přes makra (část výpočtů proveditelná v době kompilace)
- někdy není jasný bottleneck
 - přístup do paměti nemusí být zcela optimální, rychlost instrukcí obtížně odhadnutelná, schopnost překrýt výpočet a paměťové přenosy také
 - a kdo tedy zdržuje?
 - paměťové přenosy i výpočet lze benchmarkovat zvlášť
 - pak můžeme kernel rozdělit, zvětšit, zvýšit obsazenost GPU...

Měření výkonu

- samostatné paměťové přenosy
 - zakomentujeme výpočet
 - načtená data musíme nějak "použít" (často jednodušší situace u sdílené paměti)
- samostatný výpočet
 - data nemusíme načítat
 - výsledek výpočtu je však třeba uložit, aby kompilátor neodstranil výpočet
 - my ale ukládat nechceme. . .
 - uložení výsledků lze vložit do podmínky, která nikdy nebude splněna
 - u rychlých kernelů pozor na overhead jejich spuštění